DOI:
10.1007/978-3-540-44616-3_24

Document Version
Peer reviewed version

Extraction of Structured Programs from Specification Proofs

John N. Crossley,[1] Iman Poernomo[1,] * and Martin Wirsing[2]

[1] School of Computer Science and Software Engineering, Monash University, Clayton, Victoria, Australia 3168.
{jnc, ihp}@csse.monash.edu.au

[2] Institut für Informatik, Ludwig-Maximilians-Universität, Oettingenstraße 67, 80538 München, Germany.
wirsing@informatik.uni-muenchen.de

Abstract

We present a method using an extended logical system for obtaining "correct" programs from specifications written in a sublanguage of CASL. By "correct" we mean programs that satisfy their specifications. The technique we use is to extract programs from proofs in formal logic by techniques due to Curry and Howard. The logical calculus, however, has the novel feature that as well as the conventional logical rules it includes structural rules corresponding to the standard ways of modifying specifications: translating (renaming), taking unions of specifications and hiding signatures. Although programs extracted by the Curry-Howard process can be very cumbersome, we use a number of simplifications that ensure that the programs extracted are in a language close to a standard high-level programming language. We use this to produce an executable refinement of a given specification and we then provide a method for producing a program module which respects the original structure of the specification as much as possible. Throughout the paper we demonstrate the technique with a simple example.

1 Introduction

One of the most exciting applications of formal specifications is in the formal development of programs. By gradually refining a high-level specification one eventually obtains a low-level "program" or "executable specification" as in [14], [15]. If each refinement step can be proved correct, then the resulting program is guaranteed to satisfy the original specification. In this paper instead of proving the correctness of a refinement step *a posteriori*, we show how we can construct refinements from proofs in a way similar to that in which programs are extracted from proofs in mathematical logic (see [3], [6], [2], [11], [1]). As our framework we use a subset of the algebraic specification language *CASL* [19] that supports structured algebraic specifications with first-order axioms and structuring mechanisms for unions of specifications, translating, and hiding symbols from the signature of a specification. As programs we consider executable specifications where function symbols are specified by means of terms from a simply typed lambda calculus.[1]

We choose the simple notion of model inclusion for refinement:[2] a specification SP_1 is a *refinement* of a specification SP (written $\mathrm{SP} \rightsquigarrow \mathrm{SP\_1}$) if all models of SP_1 (restricted

* Research partly supported by ARC grant A 49230989.

[1] We shall call these "lambda-terms" for brevity.

[2] Note that, in combination with the structuring mechanisms, this simple notion of refinement is sufficient for expressing all the interesting notions of refinement of algebraic specifications provided one interprets the equality predicate symbol as a congruence relation instead of the standard equality (see e.g. [17]).

to sig(SP_1))are also models of SP. In the first step we derive a simply typed lambda-term e for each function symbol f of a specification SP by extracting lambda-terms from proofs of the axioms of SP over another data structure specification, say SP_0 to obtain SP_1 extending SP_0 by definitions of the form $f = e$. SP_1 is, by construction, a correct refinement of SP and, if SP_0 is executable, then SP_1 is also executable and we are done. Otherwise we repeat the process.[3]

The new contributions of this paper to the development of specifications are as follows. As far as we know, ours is the first approach (building on our earlier [18]) using program-extraction from (formal) proofs in the area of structured algebraic specifications. Moreover it enhances the program extraction techniques already developed for first-order predicate calculus by methods for dealing with structural rules. A further advantage of our approach is that by the extraction techniques studied in [1], [11] and [2], the programs that are automatically extracted are close to those a human developer would have written and the structure of the specification is mirrored in the dependencies of the module extracted. The only similar approach we know of is that of Smith [16] in the *SpecWare* system. He uses similar techniques to construct specification morphisms. Our technique differs from his in the specification-building operations and in the program-extraction technique.

The paper is organized as follows: In § 2, we introduce the specification language and introduce our example. which is developed throughout the paper. § 3 gives the background from mathematical logic, presenting a sound and complete proof-system for properties of structured specifications in constructive first-order logic. § 4 studies Curry-Howard reductions and strong normalization. and we present our method of program extraction and the transformation of the extracted programs to a "human-readable" form.

2 Structured specifications

In writing large specifications it is convenient to design specifications in a structural and modular fashion by combining and modifying smaller specifications. This helps us to master the complexity arising from a large number of function symbols and axioms.

We employ three specification-building operations from CASL [19]. A basic specification is of the form $\langle \Sigma, Ax \rangle$, where Σ is a signature consisting of a set of sorts (i.e. names for carrier sets), a set F of $S^* \to S$-sorted function symbols and a set P of $S^* \times S$-sorted predicate symbols. Ax is a set of Σ-formulae. Each such formula is a Harrop formula (see § 4) sometimes of the form $f(x_1, \ldots, x_n) = e$ where e is a λ-expression. (For the final syntax see § 4.)

The specification-building operations for constructing specifications from basic ones are: *translation*, *union* and *hiding*.[4]

As the concrete syntax for our examples we use a subset of the specification language that admits all the above constructs together with the syntax of simply typed lambda-calculus.[5] We assume that all our specifications include the appropriate axioms for equality (see

[3] Only a finite number of steps will be necessary.

[4] In [18] we used *SPECTRUM* instead of *CASL*, *building the sum* of two specifications instead of *union* and *export* instead of *hiding* and we wrote 1. $\rho \bullet$ SP for the translation of SP by ρ, where ρ is a symbol mapping, 2. SP_1 + SP_2 for the sum of the two specifications and 3. $\text{SP}\|_\Sigma$ for hiding the symbols in Σ from the signature, where Σ is a symbol list.

[5] In our language we do not treat subsorts and we assume that all functions are total.

Logical Rules

Introduction Rules

$$\frac{}{\{x:A\} \vdash_{\langle \mathtt{sig}(A),\ \emptyset\rangle} ass(A,x) : A}\ (\text{Ass I}) \qquad \frac{}{\emptyset \vdash_{\langle \Sigma, Ax\rangle} ax(\langle \Sigma, Ax\rangle, x) : A}\ (\text{Ax I})$$

if $\{x:A\} \in Ax$

$$\frac{\Gamma \bar{\cup} \{x:A\} \vdash_{\text{SP}} d : B}{\Gamma \vdash_{\text{SP}} \lambda x:A.d : (A \to B)}\ (\to \text{I}) \qquad \frac{\Gamma_1 \vdash_{\text{SP\_1}} d : A \quad \Gamma_2 \vdash_{\text{SP\_2}} e : B}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{SP\_1}} \textbf{and}\ \text{SP\_2}\ \langle d, e\rangle : (A \wedge B)}\ (\wedge \text{I})$$

$$\frac{\Gamma \vdash_{\text{SP}} d : A}{\Gamma \vdash_{\text{SP}} \langle \pi_1, d\rangle : (A \vee B)}\ (\vee_1 \text{I}) \qquad \frac{\Gamma \vdash_{\text{SP}} e : B}{\Gamma \vdash_{\text{SP}} \langle \pi_2, e\rangle : A \vee B}\ (\vee_2 \text{I})$$

$$\frac{\Gamma \vdash_{\text{SP}} d : A}{\Gamma \vdash_{\text{SP}} \lambda x:s.d : \forall x:s \bullet A}\ (\forall \text{I}) \qquad \frac{\Gamma \vdash_{\text{SP}} d : A[t/x]}{\Gamma \vdash_{\text{SP}} (t,d) : \exists x:s \bullet A}\ (\exists \text{I})$$

if $x \notin fv(B)$ for every $b : B \in \Gamma$
where x is of sort s

Elimination Rules

$$\frac{\Gamma_1 \vdash_{\text{SP\_1}} d : (A \to B) \quad \Gamma_2 \vdash_{\text{SP\_2}} r : A}{\Gamma_1 \cup \Gamma_2 \vdash_{\text{SP\_1}} \textbf{and}\ \text{SP\_2}\ (dr) : B}\ (\to \text{E}) \qquad \frac{\Gamma \vdash_{\text{SP}} d : (A_1 \wedge A_2)}{\Gamma \vdash_{\text{SP}} \pi_i(d) : A_i}\ (\wedge \text{E})$$

$$\frac{\Gamma \vdash_{\text{SP}} d : \forall x:s \bullet A}{\Gamma \vdash_{\text{SP}} dt : A[t/x]}\ (\forall \text{E}) \qquad \frac{\Gamma \vdash_{\text{SP}} d : \bot}{\Gamma \vdash_{\text{SP}} dA : A}\ (\bot \text{E})$$

$$\frac{\Gamma_1 \bar{\cup} \{x:A\} \vdash_{\text{SP\_1}} d : C \quad \Gamma_2 \bar{\cup} \{y:B\} \vdash_{\text{SP\_2}} e : C \quad \Gamma \vdash_{\text{SP}} f : (A \vee B)}{\Gamma^* \vdash_{\text{SP}*} \textbf{case}(x:A.d : C, y:B.e : C, f : (A \vee B)) : C}\ (\vee \text{E})$$

where $\Gamma^* = (\Gamma_1 \cup \Gamma_2 \cup \Gamma) \bar{\cup} \{x:A\} \bar{\cup} \{y:B\}$ and $\text{SP}* = \text{SP\_1}$ **and** SP_2 **and** SP

$$\frac{\Gamma_1 \vdash_{\text{SP\_1}} d : \exists x:s \bullet A \quad \Gamma_2 \bar{\cup} \{y:A[z/x]\} \vdash_{\text{SP\_2}} e : C}{(\Gamma_1 \cup \Gamma_2) \vdash_{\text{SP\_1}} \textbf{and}\ \text{SP\_2}\ \textbf{select}(z:s.y:A[z/x].e : C, d : \exists x:s \bullet A) : C}\ (\exists \text{E})$$

Structural Rules

$$\frac{\Gamma \vdash_{\text{SP}} d : A}{\rho'(\Gamma) \vdash_{\text{SP}} \textbf{with}\ _{\rho}\ \rho \bullet d : \rho'(A)}\ (\text{ren}) \qquad \frac{\Gamma \vdash_{\text{SP}} d : A}{\Gamma \vdash_{\text{SP}} \textbf{hide}\ _{\Sigma}\ d\ \textbf{hide}\ \Sigma : A}\ (\text{hide})$$

Γ and A may contain hidden symbols.
ρ' is an extension of ρ such that the hidden symbols are consistently translated (see e.g. [18])

if Γ is a $\mathtt{sig}(\text{SP} - \Sigma)$-context and A is a $\mathtt{sig}(\text{SP} - \Sigma)$-formula

$$\frac{\Gamma \vdash_{\text{SP\_1}} d : A}{\Gamma \vdash_{\text{SP\_1}} \textbf{and}\ \text{SP\_2}\ union_1(d,\ \text{SP\_2}) : A}\ (\text{union}_1) \qquad \frac{\Gamma \vdash_{\text{SP\_2}} d : A}{\Gamma \vdash_{\text{SP\_1}} \textbf{and}\ \text{SP\_2}\ union_2(d,\ \text{SP\_1}) : A}\ (\text{union}_2)$$

Fig. 1. Logical and Structural Rules

below), but we restrict the other axioms to be Harrop formulae.[6]

In *CASL translation* is written SP **with** ρ, where ρ is a symbol mapping,
union is written[7] SP_1 **and** SP_2
and *hiding* is written SP **hide** Σ, where Σ is a symbol list.

Note that many of the other common specification operators (extension, revealing, and local specifications) used in *CASL* can be constructed from these three operators.

Example. We use the following example throughout the paper to illustrate our method of program extraction from a structured specification. Consider the three specifications NAT_A, NAT_B and NAT_C in Fig 3. (We shall eventually show how we can obtain a program for c in NAT_C.) First we unSkolemize (see Theorem 3, § 3) the axiom for c: $\exists y : Nat \bullet y \geq s(s(s(s(0))))$. We prove this formula constructively, using the other two specifications (NAT_A and NAT_B), and then extract a program for y which computes c. From this, we can produce an executable specification, and also a program module corresponding to it, which is a refinement of NAT_C.

3 The formal calculus

We extend the Curry-Howard terms, or proof-terms, for a logical calculus for structured algebraic specifications based on classical logic, introduced in[18], to one based on constructive logic.[8,9] There are two reasons for doing this. First we can extract programs directly from the proof-terms, and secondly it allows us to make further extensions which we describe in § 4.

We use the syntax of *CASL* with logical connectives[10] $\bot$ (*falsum*), $\wedge$, $\vee$, $\rightarrow$, $\forall$ and $\exists$. The system uses sequents of the form $\Gamma \vdash_{\mathrm{SP}} d : A$ where Γ is a context, SP is a specification,[11] and d is a proof-term that gives a precise representation (see remark 5, below) of the derivation of A from the assumptions from Γ in the environment of the specification SP. From a type-theoretic point of view A is the type[12] of d. Recall that the axioms in SP may be used in addition to those from Γ to prove the formula A whose signature is (contained in) SP. As in [18] we have a set of rules. We present these in natural deduction style (see [7] or [18]). The rules for the basic system are of two kinds, logical and structural,[13] see Fig.1.

---

[6] For the definition of Harrop formulae and a discussion of their rôle see § 4. The restriction is not significant in practice and in particular is no restriction at all in the case of algebraic specifications with purely universal axioms.

[7] In particular, we use the extension "SP **then** Σ, Ax" as shorthand notation for "SP **and** $\langle \mathtt{sig}(\mathrm{SP}) \cup \Sigma, Ax \rangle$".

[8] Constructive (or Intuitionistic) logic is not commonly used in formal mathematical proofs, but in fact most proofs used in Computer Science to produce programs are either constructive or can easily be made so.

[9] Indeed, for proofs of formulae of the form $\forall x : s; \exists y : s' \bullet A(x, y)$, where A is quantifier-free, and contains no free variables other than x and y, *any* classical proof can be transformed into a constructive one (see Kleene [13] or Schwichtenberg [3]).

[10] We need all of these connectives because e.g. the De Morgan laws $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$ do not hold in general in constructive logic. Negation, $\neg A$, is an abbreviation for $(A \rightarrow \bot)$, and $(A \leftrightarrow B)$ is regarded as an abbreviation for $((A \rightarrow B) \wedge (B \rightarrow A))$.

[11] Again, as in [18], SP is considered as an equivalence class *modulo* a simple decidable equivalence relation on specifications. (See [18], §2 for more details.)

[12] This type will therefore be non-empty when A has a proof.

[13] **Remarks on the rules.** 1. For the rules $(\vee_i I)$, B must be a $\mathtt{sig}$(SP)-formula in $(\vee_1 I)$ and A must be a $\mathtt{sig}$(SP)-formula in $(\vee_2 I)$.
2. As usual $A[t/x]$ denotes the result of substituting t for all free occurrences of x in A subject to avoiding clashes of variables; and, in the rules $(\forall I)$ and $(\exists E)$, the individual variable z must not be free in C nor in any uncancelled premise.
3. The rule $(\vee E)$ is most easily understood by its analogy to proof by cases. If we have a proof of C from A

spec NAT_0 =
sorts
Nat
ops $0 : Nat; s : Nat \to Nat; + : Nat \times Nat \to Nat$
preds
$\geq : Nat \times Nat$
axioms
$Nat\_0.1 : \forall x : Nat \bullet x + 0 = x$ $Nat\_0.2 : \forall x : Nat; \forall y : Nat \bullet x + s(y) = s(x + y)$
$Nat\_0.3 : \forall x : Nat \bullet x \geq 0$ $Nat\_0.4 : \forall x : Nat; \forall y : Nat \bullet x + y = y + x$
$Nat\_0.5 : \forall x : Nat \bullet s(x) \geq x$
$Nat\_0.6 : \forall x : Nat; \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet x \geq v \wedge y \geq w \to x + y \geq v + w$
end

spec NAT_A =
NAT_0 **then**
ops $a : Nat$
axioms
$A : a \geq s(s(s(0)))$
end

spec NAT_B =
NAT_0 **then**
ops $b : Nat$
axioms
$B : b \geq s(0)$
end

spec NAT_C =
NAT_0 **then**
ops $c : Nat$
axioms
$C : c \geq s(s(s(s(0))))$
end

Fig. 2. Specifications NAT_A, NAT_B and NAT_C.

The logical rules are standard for a constructive system.[14] There are two kinds of logical rules: introduction rules and elimination rules. With the logical rules, the specification of the conclusion includes those of the premises while, for the structural rules, the change in the structure is reflected in the specification of the conclusion.[15]

Theorem 1 (Soundness and completeness). *The above system of logical and structural rules is sound and complete.*

The proof of completeness[16] proceeds as in Cengarle [5]. We use the flat (basic) normal form theorem as in [18], §2, the interpolation and compactness theorems for first-order constructive logic.

In addition to logical axioms and the axioms in the specification, we also have some implicit axioms that must be made explicit in our logical system. First the usual axioms for equality are assumed (i.e. reflexivity, symmetry, transitivity and substitutivity in both functions and predicates). Secondly, if a specification predicate is decidable, then we have the law of double negation for such predicates including equality at the base level,[17] e.g. $\forall x : s; \forall y : s \bullet (\neg\neg x = y \to x = y)$ and we note that this is a Harrop formula (see § 4).

We are now ready to prove the unSkolemized axiom for c

$$\vdash_{\text{NAT\_C}} \exists y : Nat \bullet y \geq s(s(s(s(0)))) \tag{1}$$

---

and also a proof of C from B then we get a proof of C from $A \vee B$.
4. Likewise in ($\exists$ E), if we have a proof of $\exists x : s \bullet A$ and a proof of C from a proof of A with free variable y, then we can get a proof of C.
5. As in [18] § 3, we assume that there are functions for(d), sp(d) and con(d) from proof-terms to formulae, specifications and contexts, respectively, such that for(d) is the formula proved, sp(d) = SP and con(d) is the current context. We call $sp(d)$ the *associated specification* of d.

[14] We use slight variants of the ones found in [7], but these can readily be replaced by those of any other natural deduction system.

[15] A *proof* is, as usual, a tree whose leaves are axioms or assumptions and where suceeding nodes are obtained by the rules.

[16] This logic is constructive. The same result also holds for classical logic when the law of double negation is added.

[17] Higher order equality may be a defined predicate. In such a case it may be necessary to prove the equality axioms at this higher-order level within the system.

using our calculus. We may use the axioms from NAT_C that do not involve c.
We start with the NAT_A and NAT_B axioms:

$$\vdash_{\text{NAT\_A}} ax(\text{NAT\_A}, A) : a \geq s(s(s(0))) \text{ and } \vdash_{\text{NAT\_B}} ax(\text{NAT\_B}, B) : b \geq s(0) \quad (2)$$

Then, by ($\wedge$-I) on the two formulae in (2), setting $q \equiv (ax(\text{NAT\_A}, A), ax(\text{NAT\_B}, B))$:

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B}} q : a \geq s(s(s(0))) \wedge b \geq s(0) \quad (3)$$

Now, we have the following axiom from NAT_0:

$$\vdash_{\text{NAT\_0}} ax(\text{NAT\_0}, Nat\_0.6) : \forall x : Nat; \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet x \geq v \wedge y \geq w \rightarrow x + y \geq v + w$$

Since NAT_C is built from NAT_0 we have the proof-term

$$\vdash_{\text{NAT\_C}} d : \forall x : Nat; \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet x \geq v \wedge y \geq w \rightarrow x + y \geq v + w$$

where $d = \text{union}_2(ax(\text{NAT\_0}, Nat\_0.6), \langle \mathtt{sig}(\text{NAT\_0} \cup \{c\}, C : c \geq s(s(s(s(0))))\rangle$. If we perform the structural rule (union_2) twice on this axiom with respect to the specifications NAT_A and NAT_B, then we shall have:

$$\dfrac{\dfrac{\vdash_{\text{NAT\_C}} d : \forall x : Nat; \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet x \geq v \wedge y \geq w \rightarrow x + y \geq v + w}{\vdash_{\text{NAT\_B } \mathbf{and} \text{ NAT\_C}} union_2(d, \text{NAT\_B}) : \forall x : Nat; \forall y : Nat; \forall w : Nat; \forall v : Nat \bullet x \geq v \wedge y \geq w \rightarrow x + y \geq v + w}}{\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} p : \forall x : Nat; \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet x \geq v \wedge y \geq w \rightarrow x + y \geq v + w}$$

where $p \equiv union_2(union_2(d, \text{NAT\_B}), \text{NAT\_C})$. We can then substitute a here, because a is in the current specification, i.e. applying $\forall$-E, with a for x, obtain:[18]

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} (union_2(union_2(ax(\text{NAT\_C}, Nat\_0.6), \text{NAT\_B}), \text{NAT\_C}))a : \forall y : Nat; \forall v : Nat; \forall w : Nat \bullet a \geq v \wedge y \geq w \rightarrow a + y \geq v + w$$

Next we apply $\forall$-E three more times, substituting b for y, $s(s(s(0)))$ for v and $s(0)$ for w. This gives:

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} r : a \geq s(s(s(0))) \wedge b \geq s(0) \rightarrow a + b \geq s(s(s(0))) + s(0) \quad (4)$$

where $r = (((((union_2(union_2(ax(\text{NAT\_C}, Nat\_0.6), \text{NAT\_B}), \text{NAT\_C}))a)b)s(s(s(0))))s(0))$.
Applying ($\rightarrow$)-E to (3) and (4), and setting $k \equiv rq$ gives:

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} k : a + b \geq s(s(s(0))) + s(0) \quad (5)$$

By the substitutivity of equality over predicates and the $+$-axioms in NAT_C, we obtain

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} k' : a + b \geq s(s(s(s(0)))) \quad (6)$$

for some term[19] k'. Then we apply ($\exists$)-I to (5) with y as witness for $a + b$ to give

$$\vdash_{\text{NAT\_A } \mathbf{and} \text{ NAT\_B } \mathbf{and} \text{ NAT\_C}} (a + b, k') : \exists y : Nat \bullet y \geq s(s(s(s(0)))) \quad (7)$$

Finally, we wish to prove equation (1) $\exists y : Nat \bullet y \geq s(s(s(s(0))))$ for the specification NAT_C, and hide the symbols a, b from the signatures of NAT_A and NAT_B, we obtain (setting NAT_C_1 = (NAT_A **and** NAT_B **and** NAT_C) **hide** a, b):

$$\vdash_{\text{NAT\_C\_1}} (a + b, k') \mathbf{\ hide\ } a, b : \exists y : Nat \bullet y \geq s(s(s(s(0)))) \quad (8)$$

This is the unSkolemized version of the axiom for c in NAT_C, as required. Later we shall show how to extract a program for c from this Curry-Howard term.

[18] Note that we do not need brackets in NAT_A **and** NAT_B **and** NAT_C because of footnote 11 above.
[19] Although k' is initially complicated because of the logical steps required, nevertheless it rapidly reduces as in § 4 below because the formulae involved are Harrop.

4 Extracting Programs from Proofs

There is a well-known map from constructive proof-terms to terms in a simply typed lambda calculus with product types which yields programs.[20] We call the [terms of] this calculus Λ. We now describe the analogous map for proof-terms extended with structural symbols.

The proof of strong normalization (see [18]) for such a lambda calculus shows us how to obtain (programs for) new functions from proof-terms for theorems of the form $\forall x : s; \exists y : s' \bullet A(x, y)$. Each proof-term can be thought of as a program (in a lambda calculus with dependent sum and product types). Thus, proof normalization, proof-term reduction and program evaluation may be considered as equivalent notions. A similar (naive) method of program extraction might be used for our proof-terms.

So, in this sense, the proof-term $(a + b, k')$ of

$$\vdash_{\text{NAT\_A}} \textbf{and } \text{NAT\_B} \textbf{ and } \text{NAT\_C}\ (a + b, k') : \exists y : Nat \bullet y \geq s(s(s(s(0)))) \tag{7}$$

can be thought of as a program. Unfortunately, such a direct method of program extraction yields awkward programs. There are two reasons for this: one concerns Harrop formulae, and the other the use of (hide).

Harrop Formulae and their Associated Reductions. First we note how systematic treatment of Harrop[21] formulae enables us to extract more realistic programs. Intuitively these formulae contain no essential use of $\exists$ or $\vee$. This means that the (sub-)terms that we associate with Harrop formulae have no "computational content" and therefore can be "deleted" from the program. Algebraic specifications will very often have Harrop formulae for their axioms.

Definition 1. *A formula is a* Harrop formula *if it is 1. an atomic formula, 2. of the form $(A \wedge B)$ where A, B are Harrop, 3. of the form $(Z \to A)$ where A (but not necessarily Z) is Harrop or 4. of the form $\forall x : s \bullet A$ where A is Harrop.*

Reduction Rules for proof-terms for Harrop formulae are very simple (see the appendix).

Logical Reductions. In addition to the proof-term reductions to be found in [18] we have some additional ones. Some arise from the new logical reductions for the additional connectives $\vee$ and $\exists$, and some from structural reductions. These are given below (but for full details see [7]).[22] Here is the reduction[23]

$$\frac{\dfrac{\vdots}{\Gamma_1 \bar{\cup} \{x : A\} \vdash_{\text{SP\_1}} d : C} \quad \dfrac{\vdots}{\Gamma_1 \bar{\cup} \{y : B\} \vdash_{\text{SP\_1}} e : C} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash_{\text{SP}} g : A}}{\Gamma \vdash_{\text{SP}} \langle \pi_1, g \rangle (A \vee B)}\ (\vee_1\text{-I})}{\Gamma_1 \cup \Gamma \vdash_{\text{SP\_1}} \textbf{and } \text{SP}\ \textbf{case}(x : A.d : C, y : B.e : C, \langle \pi_1, g \rangle : (A \vee B)) : C}\ (\vee\text{-E})$$

[20] In fact these can readily be transformed into programs in the usual programming languages (such as C++, ML, etc.).

[21] Harrop formulae are named for Ronald Harrop (see his [10]).

[22] The substitution that we need is as in [18] supplemented by substitution of the extra terms we have because of the additional logical rules. However, if we use the implementation of the reductions given in [7] for **case** and **select**, then we do not need any extra clauses.

[23] We have assumed that the specification and contexts for the two proofs of C are the same. It would be possible to admit different specifications and contexts but then there would need to be some adjustment in the reduction to ensure that the conclusion of the reduced proof had the same specification and context as the original. for $(\vee_1 I)$ immediately followed by $(\vee_1 E)$. This is because the conclusion may be used in a later inference.

This sequence of logical rules then reduces to

$$\frac{\begin{array}{c}\vdots\\ \Gamma_1 \bar{\cup} \{x : A\} \vdash_{\mathrm{SP\_1}} d : C \quad \Gamma \vdash_{\mathrm{SP}} g : A\end{array}}{\Gamma_1 \cup \Gamma \vdash_{\mathrm{SP\_1}} \mathbf{and}\ \mathrm{SP}\ d[g/x] : C}\ (\text{Ass-E})$$

Denoting "reduces to" by $\succ$, the corresponding proof-term reduction is:[24]

$$(\Gamma_1 \cup \Gamma) \vdash_{\mathrm{SP\_1}} \mathbf{and}\ \mathrm{SP}\ \mathbf{case}(x : A.d : C, y : B.e : C, \langle \pi_1, g \rangle : (A \vee B)) : C \succ (\Gamma_1 \cup \Gamma) \vdash_{\mathrm{SP\_1}} \mathbf{and}\ \mathrm{SP}\ d[g/x] : C$$

Any sequence of such reductions always terminates, that is to say, we have **strong normalization**. The proof is like that in [18] using techniques from [7] and Girard [9].

Extracting Programs from Modular Proofs. When we define our function **extract** giving programs from certain proof-terms, we must be careful of the use of (hide). For instance, the proof-term in formula (8) is obtained after applying (hide) to a and b:

$$\vdash_{\mathrm{NAT\_C\_1}} (a + b, k')\ \mathbf{hide}\ a, b : \exists y : Nat \bullet y \geq s(s(s(s(0))))$$

If we were to extract the program for this term using the extraction map as it is defined, ignoring the applications of (hide), we would obtain the illegal term $a + b$ which contains the hidden symbols a and b.[25] To avoid this we insist the proof-term be modular according to definition 2 below.[26] We say that a Curry-Howard term e *depends* on a symbol t if the *proof* associated with e contains the symbol t.

Definition 2. *A proof-term d is said to be* critical *with respect to a symbol list Σ if*
*1. d is of the form e **hide** Σ and*
2. if the term e depends on symbols that are in Σ: $\mathtt{sig}(sp(e)) \cap \Sigma \neq \emptyset$.
A proof-term is said to modular *if it contains no critical (sub-)proof-terms.*

Lemma 1 (Extraction). *There is a "forgetful" map* **extract** *from proof-terms to terms in Λ, such that, given any proof $\vdash_{\mathrm{SP}} d : A$, with the term d being modular, then* **extract**(d) *is in Λ, is well typed and is an extended realizer (see* [3] *and below) for A.*

extract(p) removes "non-computational" type information from p to extract a simple type. The definition of **extract** is given in the appendix and is the same as in [2] or [3], *modulo* the hiding, translating and union operators and the algorithms we have given above in §4 which are used in an extraction map for getting programs from proof-terms.[27] For the example above in (7) without **hide**, we have **extract**$((a + b, k')) = a + b$.

For the moment we consider a *CASL* syntax extended with product and disjoint union types in the range of functions simply for the convenience of presenting our results easily.

[24] Notice that the interaction is principally between the proof-terms $\langle \pi_1, g \rangle$ and d.

[25] Later, we shall in fact produce an executable specification by adding a definition for c as the [output of the] program extracted from this proof. For this we shall require that the program extracted contain references *only* to the visible symbols in its associated specification.

[26] Note that definition 2 below is, in fact, an inductive one associated with the definition of a proof.

[27] The reader is referred to our appendix and ([2] and [3]) for details, including how the specification is computed. These algorithms can be extended to an extraction map which extends the usual algorithm by replacing occurrences of $\rho \bullet t$ by the *application* of ρ to t (written $\rho(t)$) and replacing terms of the form $union_i(a, \mathrm{SP})$ by a prior to "deleting".

First recall that the Skolemization of a constructively proved $\forall\exists$ formula $\forall x : s; \exists y : s \bullet P(x, y)$ gives a formula, *equivalent* with respect to satisfaction, $\forall x : s \bullet P(x, f(x))$. This Skolemized formula is true for a certain function f.

In order to define our extraction procedure we need the *extended Skolemization* of any constructively provable formula A. That is to say, given a constructively proved formula A, we build an equivalent Harrop formula $P(f_A)$ which will be true for a certain function f_A.

Definition 3 (Extended Skolemization). *Given a closed formula A, we define the* extended Skolemization *of A to be the Harrop formula $Sk(A) = Sk'(A, \emptyset)$, where $Sk'(A, AV)$ is defined as follows. A unique function letter f_A is associated with each such formula A (see clause 6.). AV represents a list of application variables in A. (That is to say, the variables which will be the arguments of f_A.) If AV is $\{x_1 : s_1, ..., x_n : s_n\}$ then $f(AV)$ stands for the function application $f(x_1, ..., x_n)$.*

1. If A is Harrop, then $Sk'(A, AV) \equiv A$

2. If $A \equiv B \vee C$, then

$$Sk'(A, AV) \equiv \pi_1 f_A(AV) = \pi_1 \rightarrow (Sk'(B, AV)[\pi_2 f_A / f_B]) \wedge \pi_1 f_A(AV) = \pi_2 \rightarrow (Sk'(C, AV)[\pi_2 f_A / f_C]).$$

3. If $A \equiv B \wedge C$, then $Sk'(A, AV) \equiv Sk'(B, AV)[\pi_1 f_A / f_B] \wedge Sk'(C, AV)[\pi_2 f_A / f_C]$.

4. If $A \equiv B \rightarrow C$, then $Sk'(A, AV) \equiv Sk'(B, AV) \rightarrow Sk'(C, AV \cup \{f_B\})[f_A / f_C]$.

5. If $A \equiv \forall x : s \bullet B$, then $Sk'(A, AV) \equiv \forall x : s \bullet Sk'(B, AV \cup \{x : s\})[f_A / f_B]$.

6. If $A \equiv \exists y : s \bullet B$, then $Sk'(A, AV) \equiv Sk'(B, AV)[f_A(AV) / y : s]$.

So, for example, given a formula $A \equiv \exists y \bullet y \geq s(s(s(s(0))))$ we have $Sk(A) \equiv f_A \geq s(s(s(s(0))))$.

Definition 4 (Extended Realizer). *Given a formula A, f_A is an extended realizer of A if, and only if, $Sk(A)$ is provable.*

So for the example above, if we can prove $f_A \geq s(s(s(s(0))))$, then f_A is an extended realizer of A. When we extract a program for a proof of $\vdash_{\mathrm{SP}} d : A$, we produce an extended realizer for A. In particular, if we define $f_A = \mathbf{extract}((a + b, k'))$ we obtain an extended realizer for our example. We also have

Lemma 2. *If $\vdash A$ then there is an extended realizer f_A such that $\vdash Sk(A)$ and conversely.*

The next theorem shows that when we extract a term $\mathbf{extract}(d)$ in Λ from a proof $\vdash_{\mathrm{SP}} d : A$, the equality $f_A = \mathbf{extract}(d)$ and the formula $Sk(A)$ can be added to SP as axioms, and f_A can be added to $\mathtt{sig}(\mathrm{SP})$. This will give a larger specification which is a correct refinement of SP.

Theorem 2.[28] *Given a proof $\emptyset \vdash_{\mathrm{SP}} d : A$ such that d is modular, $e = \mathbf{extract}(d)$ and $\overrightarrow{x}$ (of types $T_1, \ldots, T_n$, respectively) is the list of all free variables in e, let f_A be as given by definition 4 and define $NewSpec(\mathrm{SP}, A, e) = \mathrm{NEW}$, say, by*

spec NEW =
{ SP **then**

[28] Note that for *CASL* specifications this theorem will hold only if A is a Harrop formula or does not contain non-Harrop subformulae of the form $B \rightarrow C$. Otherwise, f_A will have a higher order type and other new operators f_B will be included in the signatures.

ops $f_A : T \to s$
axioms $\forall \overrightarrow{x} \bullet f_A(\overrightarrow{x}) = e(\overrightarrow{x})$
$Sk(A)$ }
end

where $T = T_1 \times \ldots \times T_n$. *Then* $NewSpec(\text{SP}, A, e)$ *is a correct refinement of* SP.

Proof. Since e is an extended realizer of A, it follows that $Sk(A)$ is true when $f_A(x) = e(x)$.

Definition 5. *We define an additional constructor* **unextract** *for proof-terms. Given a extended realizer* e *of* A *in* SP *then we add* $\vdash_{\text{SP}} \mathbf{unextract}(e, \text{SP}) : A$ *to our logical calculus and define* $sp(\mathbf{unextract}(e, \text{SP})) = \text{SP}$.

unextract is needed for eliminating critical subterms from proofs by extracting intermediate programs from subproofs.
If the proof-term d is modular, then we can use theorem 2 to give us the following rule as a conservative extension to our calculus.

$$\frac{\emptyset \vdash_{\text{SP}} d : A}{\emptyset \vdash_{NewSpec(\text{SP},A,\mathbf{extract}(d))} \mathbf{unextract}(f_A, NewSpec(\text{SP}, A, \mathbf{extract}(d))) : A} \quad \text{(Sk)}$$

The resulting specification $NewSpec(\text{SP}, A, \mathbf{extract}(d))$ is a conservative extension of SP since if $NewSpec(\text{SP}, A, \mathbf{extract}(d))$ is inconsistent, then, by Theorem 6.20 of [7], so too is SP.
In the same way in addition to functions from proof-terms we can also consistently add functions given by explicit definitions but in the general case, proving consistency can be very difficult. The great advantage of the above process is that when we add a new function defined by a program obtained from a proof-term, then consistency is guaranteed.
Example (*cont.*). We extract a program from our proof of (7):

$$\frac{\vdash_{\text{NAT\_A}} \textbf{and} \ \text{NAT\_B} \ \textbf{and} \ \text{NAT\_C} \ (a+b, k') : \exists y : Nat \bullet y \geq s(s(s(s(0))))}{\emptyset \vdash_{\text{SP}} \mathbf{unextract}(f, \text{SP}) : \exists y : Nat \bullet y \geq s(s(s(s(0))))}$$

where $\text{SP} \equiv NewSpec(\text{NAT\_A} \ \textbf{and} \ \text{NAT\_B} \ \textbf{and} \ \text{NAT\_C}, \exists y : Nat \bullet y \geq s(s(s(s(0)))), \mathbf{extract}((a+b, k')))$.
By Theorem 2, SP is a refinement of NAT_A **and** NAT_B **and** NAT_C which includes the function symbol f for the program extracted from $(a+b, t)$. In *CASL* syntax, SP appears as follows:

spec SP =
{ NAT_A **and** NAT_B **and** NAT_C **then**
ops $f : Nat$
axioms $f = a + b$
$f \geq s(s(s(s(0))))$
}
end

Two axioms are given for f in SP: the equational (executable) definition and the Skolemized version of $\exists y : Nat \bullet y \geq s(s(s(s(0))))$.
Eliminating Critical Subterms. As has been noted, we are unable to use the rule (Sk) to extract a program directly from a term involving critical terms such as (8). If we *were* permitted to use (Sk) on (8), then the equational definition $f = a + b$ would be added as an axiom to the specification NAT_C. This axiom involves symbols which are

not visible in the signature and here adding such an axiom to a specification would result in a specification that was not well formed.

Critical terms occur often using program extraction because we use functions from other specifications. In order to achieve a refinement of a target specification, these functions are often hidden.

There are two methods for eliminating critical terms from a proof term. 1. uses the extraction rule (Sk) and involves introducing an extra function for each such term and 2. involves adding new assumptions which will be satisfied by any suitable specification.

Method 1. Using the **extract** and **unextract** maps given above, we can transform any proof term into one which contains no critical terms. However, in this case we shall acquire an extra function and definition or axiom. We first show how this may be done.

Lemma 3. *Given any term* $\emptyset \vdash_{\mathrm{SP}} d : A$*, there is a term* $\emptyset \vdash_{\mathrm{SP}'} \psi(d) : A$ *such that* $\psi(d)$ *contains no critical subterms and* $\mathrm{SP} \rightsquigarrow \mathrm{SP}'$.

Proof. We give a recursive definition of $\psi(d)$ using a depth-first traversal of the proof tree represented by d. Let $n(t)$ be the total number of critical subterms in the proof-term t. We define a terminating sequence of proof terms $d = d_0, ..., d_k = \psi(d)$. Given d_i, we determine d_{i+1} as follows.

Case 1. If d_i does not contain any critical subterm, then $d_{i+1} = \psi(d) = d_i$ (viz, the sequence terminates).

Case 2. Otherwise, normalize d_i to give a proof term d'. As long as d_i has no assumptions, the normalized proof term d' will contain no subterms of the form $(\lambda x : A.p)$ **hide** Σ.
1. Take the leftmost innermost critical subterm of the form $t = e$ **hide** $\Sigma : B$ in d'. That is, take the first critical subterm t in d' which does not contain any critical subterms. So, e itself contains no critical subterms.
2. Apply (Sk) to $e : B$ to yield a new program $f =$ **extract**(e). Let $\mathrm{SP} \equiv NewSpec(sp(e), B,$ **extract**$(e))$. Then, in t, replace e by **unextract**(f, SP), and replace all occurrences of Σ by $\Sigma \cup \{f\}$ to give[29] t'.
3. Replace all occurrences of t by t' in d', to give $d_{i+1} = d'[t'/t]$. The d_{i+1} has at least one less critical subterm than d_i, and proves the same theorem as d_i.
Since $n(d_{i+1}) < n(d_i)$, this process yields a k such that $n(d_k) = 0$. Then we take $\psi(d) = d_k$, which is a proof term with no critical subterms. □.

Note that $\mathrm{SP}' = \mathrm{sp}(d')$ and will be a conservatively correct refinement of SP by the definition of **extract**. The final specification will be a refinement of $\mathrm{sp}(d)$. So, as a corollary of this lemma and our (Sk) rule, we can extract a program from any proof. First, we apply the procedure outlined in this lemma to remove critical subterms. Then we apply (Sk) to extract the program.

Example (*cont.*). If we apply the procedure of Lemma 3 to (8), we obtain the proof-term

$$\vdash_{\mathrm{NAT\_C}'} \textbf{hide}_{\ a,b}\ (\textbf{unextract}(f, \mathrm{NAT\_C}'), k')\ \textbf{hide}\ a, b : \exists y : Nat \bullet y \geq s(s(s(s(0))))$$

where NAT_C′ is

$$NewSpec(\mathrm{NAT\_A}\ \textbf{and}\ \mathrm{NAT\_B}\ \textbf{and}\ \mathrm{NAT\_C}, \exists y : Nat \bullet y \geq s(s(s(s(0)))), \textbf{extract}(a + b))$$

As can be seen, this new term contains no critical subterms. If we now extract a program from this term using (Sk),we obtain the following executable specification:

[29] Note that t' proves the same theorem as t by Lemma 2 above and the definition of **unextract**, and is not a critical term.

spec NAT_C′ =
{ NAT_A **and** NAT_B **and** NAT_C **then**
ops f : *Nat*
axioms $f = a + b$
$f \geq s(s(s(s(0))))$
}
hide a, b
end

This is a conservatively correct refinement of NAT_C (and of NAT_C_1) as required.

Method 2. There is an alternative means of eliminating critical terms. This involves transforming a proof by replacing (hide) rules with (Ass-I) rules. The disadvantage of this method is that although we may have established a formal proof for a formula to which we apply *hide*, we lose this information by taking the formula proved as a new assumption. We omit the details.

Executable Refinements. We refine a specification SP_*start* to an executable specification SP_*ex*. Then SP_*ex* is said to be *executable* if every function in the signature has an equational definition in $Ax(\text{SP\_}ex)$ of the form $f = t$ (where t is a term in Λ). Recall (see Remark 5, footnote 13, above) that each valid proof-term t has an associated structured specification $sp(t)$. We use this property and our extraction rule (Sk) to produce the required executable refinement.

Theorem 3 (Executable refinements). *Given a specification* SP_*start. If every un-Skolemized axiom in* $Ax(\text{SP\_}start)$ *is constructively provable in our calculus (possibly using other specifications), then there is an executable specification* SP_*ex which is a conservatively correct refinement of* SP_*start.*

Proof. We construct a finite series of refinements $\text{SP\_}start = \text{SP\_}0 \rightsquigarrow \ldots \rightsquigarrow \text{SP\_K} = \text{SP\_}ex$. Given a specification SP_I, we obtain SP_I+1 as follows.

1. Take the first function symbol $g \in \mathtt{sig}(\text{SP\_I})$ which is not a constructor for a sort and does *not* have an executable definition in $Ax(\text{SP\_I})$.
2. Take the conjunction of all the axioms in $Ax(\text{SP})$ in which g occurs.
3. UnSkolemize this conjunction. This will produce a formula of the form $A \equiv \forall \overrightarrow{x}\ \exists y \bullet P(\overrightarrow{x}, y)$.
4. The proof of this conjunction gives a proof-term of the form $\vdash_{\text{SP\_I}} d : A$ such that its structured specification $\text{sp}(d)$ has exactly the same[30] signature as SP_I.
   Now let SP_I+1 be SP′ **with** ρ where ρ is the translation given by $f_A \mapsto g$. Then SP_I+1 is a refinement of SP′. SP_I+1 will contain an (executable) declaration of the form $\forall \overrightarrow{x}\ g(\overrightarrow{x}) = p(\overrightarrow{x})$ (where $\overrightarrow{x}$ is the list of variables in p). SP_I+1 has one less non-executable function in its signature than SP′.

Repeating the process yields a finite strict chain of refinements in which all functions of SP have executable definitions.

Note: The definitions of the functions in SP_*ex* are *relatively* executable in the sense that they may use non-executable functions in other (sub-)specifications.

Example (*cont.*) In NAT_C′ **with** ρ where ρ is the translation given by $f \mapsto c$, we have the required executable refinement of NAT_C.

Program Modules. It would be desirable to take such an executable specification and to map it to a set of programs (a module) which have a structure that mirrors the structure

[30] Note that the proof may use any axioms from SP_*i* *except* those in which g occurs.

of the executable specification. For the purposes of this paper we define a simple modular programming language with a clear semantics that supports union, translation and hiding of modules.[31]

Formally, we take a basic program module to be defined by tuples of declarations. A declaration is an equality between a unique typed identifier $f : T$ and a program $p : T$ (where the types of f and p must coincide): $f = p$

A *generic* module is a lambda abstraction from program modules to program modules. So a *non-generic* module is either a basic program module, or is formed from other non-generic modules via the operations of translating M **with** ρ, taking the union M **and** M' and hiding M **hide** Σ (where M, M' are non-generic modules, Σ is a symbol list and ρ is a symbol map).

Here is the BNF notation for modules:

$$
\begin{array}{ll}
Program & ::= \Lambda\text{-term} \\
Name & ::= \text{typed function variable} \\
Sig & ::= Name \mid Sig \times Name \\
Declaration & ::= Name \text{“}=\text{”} Program \\
ModuleContents & ::= Declaration \mid ModuleContents \times ModuleContents \\
Module & ::= ModuleContents : Spec \mid Module \textbf{ and } Module \mid \\
 & \quad\; Module \textbf{ with } \rho \mid Module \textbf{ hide } \Sigma \mid ModuleVar \\
GenericModule & ::= \lambda ModuleVar.Module \mid \lambda ModuleVar.GenericModule \\
ModuleVar & ::= Variable : Spec
\end{array}
$$

where *Spec* ranges over structured specifications, Σ ranges over symbol lists, and ρ ranges over symbol maps.

Notes: 1. A program declared in a module may contain references to functions not declared within the same module. 2. The operator ρ can be thought of as a module *adapter*, which allows for modules to be reused with different names for functions. If we were to apply any translating maps to specifications prior to making them into modules, then there would be no need to add translating to the syntax of the language.

We place the following restrictions on modules:[32] 1. Semantically, generic modules are abstractions over basic (flat) program modules only. Higher order abstraction is not permitted. 2. The type of each name-program pair in any module must be first order. Higher order types are not permitted.

Extracting Modules from Specifications. We can extract a basic module from a basic executable specification SP simply by taking all function definitions as declarations in the module. We build a structured module from a structured specification by mapping the structure building operators of the specification to the corresponding operators of the module. One problem is that the declarations may contain references to functions of specifications that are not executable. We treat these specifications as module variables, and abstract over them at the end of the process to produce a generic module. This process will give us the following theorem.

Theorem 4 (Extracting Program Modules). *Given an executable specification* SP*, there exists a (possibly generic) module* $M \equiv \lambda X_1 : \text{SP\_1}, ..., \lambda X_{\text{N}} : \text{SP\_N}.M'$ *such that* $M[U_1 : \text{SP\_1}]...[U_{\text{N}} : \text{SP\_N}]$ *is a realizer of* SP *if each* U_I *is a realizer of* SP_I *for* $I = 1, \ldots, \text{N}$.
When $\text{N} = 0$, $M \equiv M'$, *a realizer of* SP.

[31] “Programs” here means terms of our simply typed lambda calculus Λ.

[32] These restrictions could be relaxed. Note that generic modules allow us to express higher order dependence of functions on functions, but that this dependence is not “explicitly” revealed in the typing of name-program pairs.

Proof. M is the result of the recursive procedure $GetModule$ applied to SP. At the same time $GetModule$ will give us a list of variables $BV(\text{SP}) \equiv \{X_1 : \text{SP\_1}, ..., X_n : \text{SP\_N}\}$.
1. If SP is basic and executable, then $GetModule(\text{SP})$ is the module obtained by taking all declarations in SP as declarations in the module. $BV(\text{SP}) = \emptyset$. Clearly $GetModule(\text{SP})$ is a realizer of SP.
2. If SP is basic and not executable, then $GetModule(\text{SP})$ is a module variable $X : \text{SP}$. $BV(\text{SP}) = \{X : \text{SP}\}$.
3. If SP is of the form SP_1 **and** SP_2 then $GetModule(\text{SP}) = GetModule(\text{SP\_1})$ **and** $GetModule(\text{SP\_2})$. $BV(\text{SP}) = BV(\text{SP\_1}) \cup BV(\text{SP\_2})$.
4. If SP is of the form SP_1 **hide** Σ, then $GetModule(\text{SP}) = GetModule(\text{SP\_1})$ **hide** Σ. $BV(\text{SP}) = BV(\text{SP\_1})$.
5. If SP is of the form SP_1 **with** ρ, then $GetModule(\text{SP}) = GetModule(\text{SP\_1})$ **with** ρ. $BV(\text{SP}) = BV(\text{SP\_1})$.

Finally, let M be given $M \equiv \lambda X_1 : \text{SP\_1}, ..., \lambda X_n : \text{SP\_N}.GetModule(\text{SP})$ where $BV(\text{SP}) = \{X_1 : \text{SP\_1}, ..., X_n : \text{SP\_N}\}$.

The rest of the proof follows from the definition of generic modules, and the fact that if SP is a basic executable specification, then $GetModule(\text{SP})$ is obviously a realizer for SP.

If SP and all its subspecifications are executable, then $BV(\text{SP}) = \emptyset$, N = 0 and M is a non-generic module. □

Example(*cont.*). The executable specification NAT_C′ is mapped to the following module using the algorithm of Theorem 4: $GetModule(\text{NAT\_C}') =$

$$
\begin{array}{l}
\lambda X_1 : \text{NAT\_A}\ \lambda X_2 : \text{NAT\_B} \\
(\\
\quad (\ \ X_1\ \textbf{and}\ X_2\ \textbf{and} \\
\quad\ \ f = a + b \\
\quad\quad c = f \\
\quad\ \ )\ \textbf{hide}\ a, b)
\end{array}
$$

Because NAT_A and NAT_B are not executable specifications, they correspond to module variables X_1 and X_2 respectively. If we can find modules which realize these specifications, then we can instantiate this generic module, obtaining a working program which computes c. The programs for a and b are encapsulated in $GetModule(\text{NAT\_C}')$. However, the function f is not encapsulated (although its definition is). So, $c = f$ is a correct definition of c that respects the encapsulation of the two submodules.

Design Considerations. We have seen how specification building operations correspond to module building operations. The location of the (hide) rules breaks the proof up into sections which correspond to modules: once the location of the (hide) rule is fixed, the design decision has been made.In the present procedure the proof process corresponds to a module design process where fixed decisions are made with respect to the placement of the (hide) rules. The application of (hide) in a proof corresponds to a design decision about the encapsulation of the resulting modules which will be extracted. However it is possible to move the other structural rules up and down proofs (see [18]). For example, if a logical argument is reused for several different specifications, then it will probably be convenient to leave all the translations to the end of the proof. We shall also normally try to move all the translations together into one large translation.

5 Conclusion

In this paper we have described a method which combines the techniques of structured specifications and program extraction in order to produce correct programs from structured specifications. As Sannella pointed out to us after seeing [18], it is not possible to separate the structural rules for building specifications from the logical rules in a proof completely. Nevertheless it *is* possible to provide a modified proof which gives rise to program modules. These modules are principally determined by the location of the applications of hiding (or export, as it was in [18]). We have shown how this can be done and illustrated the technique by a very simple example.[33]

The Curry-Howard technique we have used gives rise, in its simplest form, to very complicated programs. However, by the heavy use of techniques dependent on Harrop formulae we are able to reduce this dramatically. We have partially implemented our system and it produces readable programs, with a highly modular structure, in *ML*, directly from proofs.

The techniques we use are readily extended to systems with induction (and therefore recursion in the programs).

We have presented our work in the context of *CASL* and a logical system which is very standard so that it will be as easy as possible to read.

References

1. Albrecht, D.W. and J.N. Crossley, Program extraction, simplified proof-terms and realizability, Technical Report no. 271, Dept. of Computer Science, Monash University, Australia, 1997.
2. Anderson, P., Representing proof transformations for program optimization, in *Proceedings of the 12th International Conference on Automated Deduction*, Nancy, France, Jun 1994, Springer-Verlag LNAI 814.
3. Berger, U. and H. Schwichtenberg, Program development by Proof Transformation, p.1-45 in *Proceedings of the NATO Advanced Study Institute on Proof and Computation*, Marktoberdorf, Germany, July 20-Aug. 1, 1993, published in cooperation with the NATO Scientific Affairs Division.
4. Broy, M. and S. Jähnichen (eds.), *KORSO: Methods, Languages, and Tools for the Construction of Correct Software, Final Report*, Lecture Notes in Computer Science 1009, Springer, Berlin, 1995.
5. Cengarle, M.V., *Formal Specifications with Higher-Order Parametrization*, Ph.D. Thesis, Ludwig-Maximilians-Universität, München, 1994.
6. Constable, R.L., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.
7. Crossley, J.N. and J.C. Shepherdson, Extracting programs from proofs by an extension of the Curry-Howard process, pp. 222-288 in *Logical Methods*, ed. J.N. Crossley, J.B. Remmel, R.A. Shore, and M.E. Sweedler, Birkhäuser, Boston (1993).
8. Gallier, J. Constructive Logics. A Tutorial on Proof-systems and Typed λ-Calculi, TCS, **110**, (1993) 249–339.
9. Girard, J.-Y., Y. Lafont and P. Taylor, *Proofs and types*, Cambridge University Press, 1989.
10. Harrop, R. Concerning formulas of the types $A \to B \vee C$, $A \to (Ex)B(x)$ in Intuitionistic Formal Systems, J. Symb. Logic, **25**, (1960), 27-32.
11. Hayashi, Susumu and Hiroshi Nakano. *PX, a computational logic*. MIT Press, Cambridge, Mass., 1988.
12. Hennicker, R., M. Wirsing and M. Bidoit, Proof systems for structured specifications with observability operators, TCS **173**, (1997), 393-443.
13. Kleene, S.C. *Introduction to Metamathematics*, North-Holland, Amsterdam (1952).
14. Sannella, D.T. and A. Tarlecki, Toward formal development of programs from algebraic specifications: Implementations revisited. Acta Informatica **25**, (1988), 233-281.
15. Santen, T, F. Kammüller, S. Jähnichen, M. Beyer, *Formalization of Algebraic Specification in the Development language DEVA* , p. 223-238 in [4].
16. Smith, D.R. *Constructing Specification Morphisms*, J. Symbolic Computation (1993), **15**, 571-606.
17. Wirsing, M. and M. Broy: A modular framework for algebraic specification and implementation. In: J. Diaz, F. Orejas (eds.): TAPSOFT 89, Lecture Notes in Computer Science, 351, Berlin, Springer, 1989.
18. M. Wirsing, J.N. Crossley and H. Peterreins, Proof normalization of structured algebraic specifications is convergent, in J Fiaderio (ed), *Proceedings of the Twelfth International Workshop on Recent Trends in Algebraic Development Techniques*, Lecture Notes in Computer Science **1589**, Springer-Verlag, Berlin, 1999, p. 322-337.
19. CoFI Language Design Task Group, *CASL - The CoFI Algebraic Specification Language - Summary, version 1.0*, 22 July 1999, available at `http://www.dcs.ed.ac.uk/home/dts/CoFI/Documents/CASL/Summary/index.html`

[33] In a planned journal article we shall give much more substantial examples.

Appendix: Reduction Rules and Extracting Programs

Here we briefly supplement the description of the extraction of programs from proof-terms of § 4. In obtaining programs there are two stages that need to be distinguished. One is the normalization process of the logical (and structural) calculus which uses the logical and structural reductions and proceeds in a standard way as in [7] and [1].) This produces a simplified proof-term. After this there is the actual extraction of the program.

Normalizing reductions There are two types of normalizing reductions: those for logical rules which follow the traditional pattern initiated by Curry and Howard (see, e.g. [7]), and those for structural rules initiated by us in [18]. We also briefly mention the permutation of rules mentioned at the end of § 4. (We omit brackets and the intial $\Gamma \vdash_{\mathrm{SP}}$ as Γ and SP can be found by applying the function sp to the term. (See footnote 11, remark 5.)

Logical Reductions

For each connective we can reduce when an introduction is immediately followed by an elimination. In § 4 we explicitly showed how ($\vee_1$-I) followed by ($\vee$-E) reduces. The corresponding reductions for the other connectives are as follows:[34]

$\wedge$	$\pi_i(\langle a_1 : A_1, a_2 : A_2\rangle : (A_1 \wedge A_2)) \succ a_i : A_i$ for $i \in \{1,2\}$.
$\rightarrow$	$(\lambda x : A.d : (A \rightarrow B))(a : A) \succ d[a/x] : B$
$\forall$	$(\lambda x : s.d : (\forall x : s \bullet B))(a : s) \succ d[a/x] : B[a/x]$
$\exists$	$\mathbf{select}(z : s.y : A[z/x].e : C, d : \exists x : s \bullet A) : C \succ e[\pi_1(d)/z][\pi_2(d)/y] : C$

Structural reductions

These are commonsense reductions: two translations can be consolidated into one; and enriching by Σ and then hiding Σ, or *vice versa*, reduces to a triviality.

Composition	$(\rho_2 \bullet (\rho_1 \bullet d : \rho_1(A))) : \rho_2(\rho_1(A)) \succ \rho\prime \bullet d : \rho\prime(A)$ where $\rho\prime = \rho_2 \circ \rho_1$.

For $i \in \{1,2\}$ and $\Sigma \subseteq Sig(\mathrm{SP})$:

hide/enrich$_i$	$enrich_i((d : A \ \mathbf{hide}\ \Sigma) : A, \mathrm{SP}) : A \succ d : A$
enrich$_i$/hide	$enrich_i(d : A, \mathrm{SP})\ \mathbf{hide}\ \mathtt{sig}(\mathrm{SP}) : A \succ d : A$

Program Extraction First we define the types for our lambda calculus by:

$$\tau, \tau_1, \tau_2, \tau_3 ::= s \mid \tau_1 \times \tau_2 \mid \tau_1 | \tau_2 \mid \mathrm{H}$$

where s ranges over the set of sorts.
Terms of Λ are given by the following grammar

$$t_1, t_2, t_3 ::= \alpha \mid x \mid () \mid \pi_1 t_1 \mid \pi_2 t_2 \mid \lambda x : \tau.t_1 \mid \langle \pi_1, t_1\rangle \mid \langle \pi_2, t_1\rangle \mid (t_1, t_2) \mid t_1 t_2 \mid \mathbf{case}(x : \tau_1.t_1, y : \tau_2.t_2, t_3) \mid \mathbf{select}(x : \tau_1.y : \tau_2.t_1, t_2)$$

where α ranges over terms whose sort is in s and x ranges over variable names.

[34] The explicit formulations of the reductions in the logical proofs are given in, for example, [7].

The type formation rules for Λ are as follows.

$$\frac{}{\vdash () : \mathrm{H}} \qquad \frac{}{\alpha : s} \qquad \frac{}{x : s \vdash x : s}$$

where α is a term of sort s.

$$\frac{\Gamma, x : \tau_1 \vdash t}{\Gamma \vdash \lambda x : \tau_1 . t} \qquad \frac{\Gamma_1 \vdash d : \tau_1 \to \tau_2 \quad \Gamma_2 \vdash r : \tau_1}{\Gamma_1, \Gamma_2 \vdash dr : \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash (\pi_i t) : \tau_i \quad i \in \{1,2\}} \qquad \frac{\Gamma_1 \vdash t_1 : \tau_1 \quad \Gamma_2 \vdash t_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (t_1, t_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \langle \pi_1, t \rangle : \tau_1 | \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \langle \pi_2, t \rangle : \tau_1 | \tau_2}$$

$$\frac{\Gamma_1, x : \tau_1 \vdash d : \tau \quad \Gamma_2, y : \tau_2 \vdash e : \tau \quad \Gamma_3 \vdash f : \tau_1 | \tau_2}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathbf{case}(x : \tau_1 . d, y : \tau_2 . e, f) : \tau}$$

$$\frac{\Gamma_1, x : \tau_1, y : \tau_2 \vdash e : \tau \quad \Gamma_2 \vdash f : \tau_1 \times \tau_2}{\Gamma_1, \Gamma_2 \vdash \mathbf{select}(x : \tau_1 . y : \tau_2 . e, f) : \tau}$$

The operational semantics of Λ is given below. (These are the usual reduction rules for simply typed lambda calculus augmented by product and union types).

$$(\lambda x : \tau_1 . d) r \longrightarrow d[r/x]$$

$$\mathbf{case}(x_1 : \tau_1 . d_1, x_2 : \tau_2 . d_2, (\pi_i, f)) \longrightarrow d_i[f/x_i] \quad \text{for } i \in \{1,2\}$$

$$\mathbf{select}(x : \tau_1 . y : \tau_2 . d, (a, b)) \longrightarrow d[a/x][b/y]$$

$$\pi_i(t_1, t_2) \longrightarrow t_i \quad \text{for } i \in \{1,2\}$$

We define a map ϕ over formulae of our logical system to types of Λ. Given a formula F, $\phi(F)$ will give us the (computational) type of the program extracted from the proof of F. ϕ is defined by cases.

If A is Harrop, then $\phi(A) = \mathrm{H}$, otherwise, ϕ is defined as below.

$$\phi(\exists x : s \bullet A) = \begin{cases} s & \text{if } Harrop(A) \\ s \times \phi(A) & \text{otherwise} \end{cases}$$

$$\phi(\forall x : s \bullet A) = \lambda x : s . \phi(A)$$

$$\phi(A \to B) = \begin{cases} \phi(B) & \text{if } Harrop(A) \\ \phi(A) \to \phi(B) & \text{otherwise} \end{cases}$$

$$\phi(A \wedge B) = \begin{cases} \phi(A) & \text{if } Harrop(B) \\ \phi(B) & \text{if } Harrop(A) \\ \phi(A) \times \phi(B) & \text{otherwise} \end{cases}$$

$$\phi(A \vee B) = \phi(A) | \phi(B)$$

$$\mathbf{extract}(ass(A,x):A) = \begin{cases} () & \text{if Harrop(A)} \\ x & \text{otherwise} \end{cases}$$

$$\mathbf{extract}(ax(\langle \Sigma, Ax\rangle, x):A) = ()$$

$$\mathbf{extract}((\lambda x:A.d):A\to B) = \begin{cases} \mathbf{extract}(d) & \text{if } Harrop(A) \\ () & \text{if } Harrop(A\to B) \\ \lambda x:\phi(A).\mathbf{extract}(d) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}((a,b):A\wedge B) = \begin{cases} \mathbf{extract}(a) & \text{if } Harrop(A) \text{ and not } Harrop(B) \\ \mathbf{extract}(b) & \text{if } Harrop(B) \text{ and not } Harrop(A) \\ () & \text{if } Harrop(A\wedge B) \\ (\mathbf{extract}(a),\mathbf{extract}(b)) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}((\lambda x:s.d):\forall x:SB) = \begin{cases} () & \text{if } Harrop(\forall x:S\bullet B) \\ \lambda x:s.\mathbf{extract}(d) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}((t,d):\exists x:s\bullet A) = \begin{cases} t & \text{if } Harrop(A) \\ (t,\mathbf{extract}(d)) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}(\langle \pi_1, d:A_1\rangle : A_1\vee A_2) = \langle \pi_1, \mathbf{extract}(d)\rangle$$

$$\mathbf{extract}(\langle \pi_2, d:A_1\rangle : A_1\vee A_2) = \langle \pi_2, \mathbf{extract}(d)\rangle$$

$$\mathbf{extract}((d:A\to B)(r:A):B) = \begin{cases} \mathbf{extract}(d) & \text{if } Harrop(A) \\ () & \text{if } Harrop(A\to B) \\ & \text{otherwise} \end{cases}$$

$$\mathbf{extract}((d:\forall x:s\bullet B(x))(r:s):B[r/x]) = \begin{cases} () & \text{if } Harrop(B[d/x]) \\ (\mathbf{extract}(d)r) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}(\mathbf{case}(x:A.d:C,\ y:B.e:C, f:(A\vee B)):C = \begin{cases} () & \text{if } Harrop(C) \\ \mathbf{case}(\mathbf{extract}(x).\mathbf{extract}(d), \\ \mathbf{extract}(y).\mathbf{extract}(e),\mathbf{extract}(f)) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}(\mathbf{select}(z:s.y:A[z/x].e:C,\ d:\exists x:s.A):C) = \begin{cases} () & \text{if } Harrop(C) \\ ((\lambda x:s.\mathbf{extract}(e))\mathbf{extract}(d)) & \text{if } Harrop(A) \\ \mathbf{select}(z:s.\mathbf{extract}(y).\mathbf{extract}(e), \\ \mathbf{extract}(d)) & \text{otherwise} \end{cases}$$

$$\mathbf{extract}(union_i(d,SP)) = \mathbf{extract}(d)$$
$$\mathbf{extract}(\rho\bullet d) = \mathbf{extract}(\rho(d))$$
$$\mathbf{extract}(d\ \mathbf{hide}\ l) = \mathbf{extract}(d)$$

Fig. 3. The definition of **extract**.

The definition for **extract** is in Fig. 3. It assumes the proof-terms are *modular*.
Note that the final specification and the list of assumptions can easily be computed from the original proof-term.